# Antialiasing Implemented Using Streaming SIMD Extensions

## Version 2.1

## 01/99

Order Number: 243640-002

# Table of Contents

## Revision History

| Revision | Revision History | Date |
|---|---|---|
| 2.1 | FCS revision. | 01/99 |

## References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *3D Computer Graphics*, Alan Watt, second edition, pg 353 – 370, (Copyright 1993, Addison-Wesley Publishing Company, Inc.)
2. *Computer Graphics*, Foley, Van Dam, Feiner, Hughes, Second edition in C, pg 617 – 647, (Copyright 1996, Addison-Wesley Publishing Company, Inc.)

# 1 Introduction

The Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide floating-point single-instruction, multiple-data (SIMD) instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio. This application note discusses how to use the Streaming SIMD Extensions to achieve better performance in a computer graphics antialiasing algorithm, and includes examples of code that exploit the Streaming SIMD Extensions. This application note describes a commonly used technique of the antialiasing algorithm, *Supersampling with Bilinear filter*ing (see references [1] and [2]).

# 2 Antialising Algorithm

Aliasing in computer graphics results from the creation of display images using a point sampling method along with a regular sampling process in a space domain. One of the major aliasing artifacts is the "jagged" appearance along the edges of the displayed image. Antialiasing is a software technique for diminishing the visual effects of aliasing. Postfiltering, also known as supersampling, is the antialiasing technique discussed in this paper.

The supersampling algorithm can be divided into two steps, sampling and filtering:

1. The continuous image domain (or higher resolution raw image data) is sampled at *n* times the display image resolution. For example, suppose the display resolution is 400x600. Sampling at two times the width and height of the display resolution would yield 800x1200 superpixels.
2. Each pixel value of the display image is calculated from the pixel values of a set of *n***n* corresponding superpixles by certain filter algorithms. Filtering is a technique used to combine samples to compute a pixel value. Filtering methods differ in the value of *n*. This paper concentrates on the case of *n* = 2, or *bilinear filtering*.

The processes of sampling and filtering are shown in Figure 1. The outputs of sampling are rectangular superpixels and a UV lookup table. Superpixels are generated by sampling the source image at two times the width and height of the display image resolution. Therefore, the dimensions of the superpixels are (2*image_width)*(2*image_height). In the implementation presented in this paper, the pixel values of the superpixels are stored in an Array of Structures (AoS), defined as follows:

```
Typedef struct pixel{
unsigned char red, green,blue,alpha;
}Pixel;
Pixel SuperPixel[size]
```

1? where size = (2*image_width)*(2*image_height);

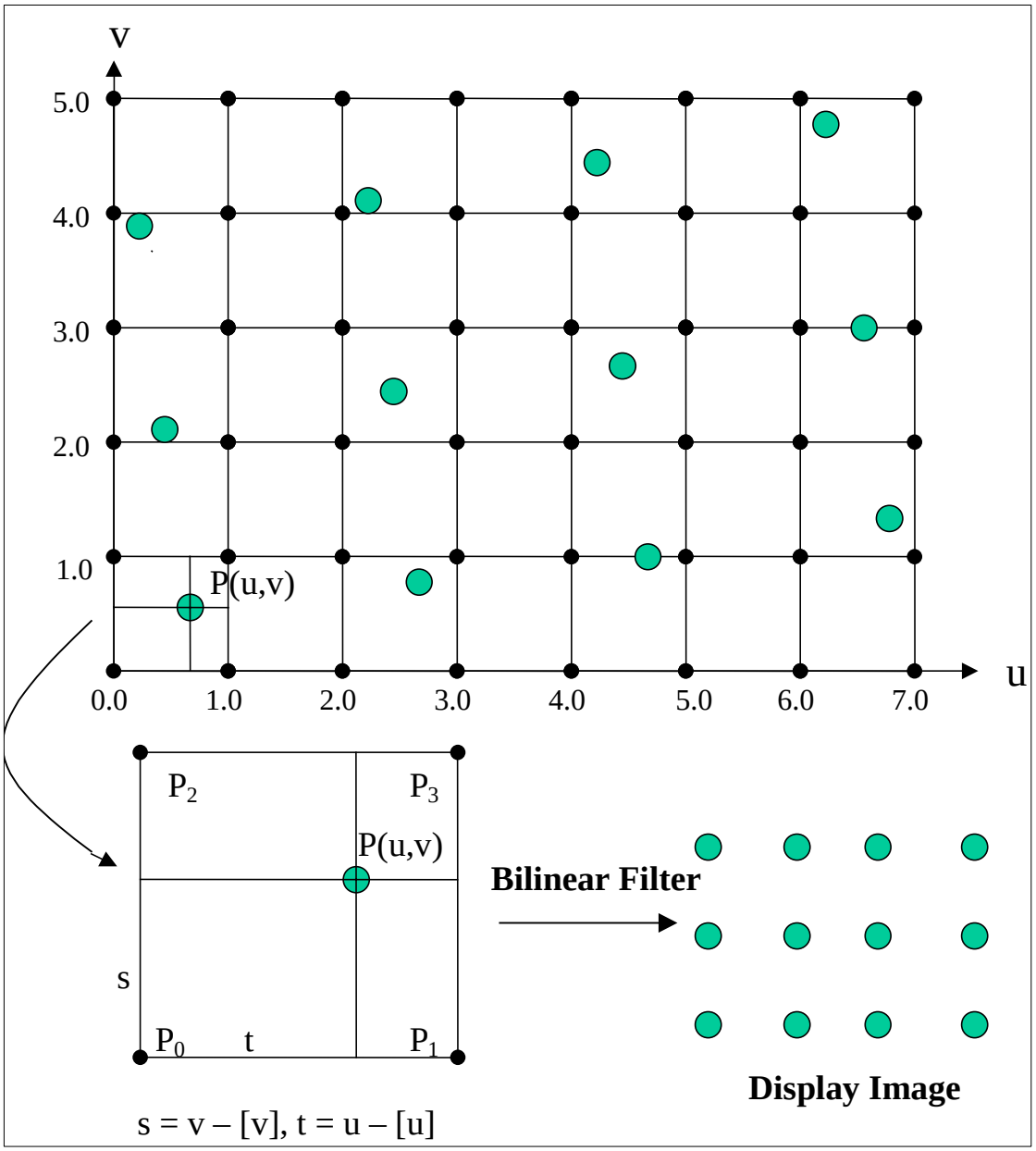


**Figure 1. Sampling and Bilinear Filtering.** The picture above shows that a rectangular superpixels are generated by sampling the source image at two times of the width and height of the display image resolution. Each pixel of the display image is mapped to a point P(u,v) in the UV domain of the source image. The picture below shows that pixel values of the display image are calculated from the pixel values of four adjacent superpixels.

As shown in Figure 1, each pixel of the display image is mapped to a point P(u,v) in the UV domain of the source image. The UV lookup table is used to store the u and v coordinates of all the pixels of the display image. The UV domain has been normalized so that the horizontal/vertical distance of two consecutive superpixels is equal to one. In the implementation, the UV lookup table is stored as an array of floats, as shown below:

```
float  uv_table[table_size];
```

where table_size = 2*image_width*image_height

For example, (uv_table[0], uv_table[1]) in the array are the u and v coordinates of the first pixel of the display image in the UV domain of the source image.

The method of sampling depends on the type of source image. This application focuses on the implementation of the second step of the Antialiasing algorithm, *Bilinear filtering.*

Filtering is a process of computing the pixel values of a display image by calculating a weighted average of the pixel values of adjacent superpixels. For bilinear filtering, the weighted average is computed as a bilinear interpolation to the four nearest adjacent superpixels. Suppose a pixel of the display image is mapped to a point *P(u,v)* in the UV domain of the supersampling space, and $P_0$, $P_1$, $P_2$ and $P_3$ are the nearest four adjacent superpixels of the point *P(u,v)* (refer to Figure 1). The integer parts of *u* and *v* are used to find the locations of $P_0$, $P_1$, $P_2$ and $P_3$. For example, if [u] and [v] are the integer parts of *u* and *v*, then ([u],[v]), ([u+1],[v]), ([u],[v+1]) and ([u+1],[v+1]) are coordinates of the four superpixels in the supersampling space. The fractional parts of *u* and *v* are the horizontal and vertical distances of *P* to $P_0$. These distances serve as 'weight factors' in the interpolation: closer superpixels have more weight.

Let *t* and *s* be the weight factors of the interpolation, then

t = u – [u]

s = v – [v]

where [u] and [v] are the integer parts of *u* and *v*, 0.0 <= t <= 1.0 and 0.0 <= s <= 1.0

Let $(R_0,G_0,B_0,A_0)$, $(R_1,G_1,B_1,A_1)$, $(R_2,G_2,B_2,A_2)$ and $(R_3,G_3,B_3,A_3)$ be the 4-component pixel values of the superpixels $P_0$, $P_1$, $P_2$ and $P_3$. The RGBA pixel values of the sampling point *P* are calculated by a bilinear interpolation to $(R_0,G_0,B_0,A_0)$, $(R_1,G_1,B_1,A_1)$, $(R_2,G_2,B_2,A_2)$ and $(R_3,G_3,B_3,A_3)$ respectively:

$$R = (1.0 - s) * [(1.0 - t) * \boldsymbol{R_0} + t * \boldsymbol{R_1}] + s * [(1.0 - t) * \boldsymbol{R_2} + t * \boldsymbol{R_3}] \quad \text{(I)}$$

$$G = (1.0 - s) * [(1.0 - t) * \boldsymbol{G_0} + t * \boldsymbol{G_1}] + s * [(1.0 - t) * \boldsymbol{G_2} + t * \boldsymbol{G_3}] \quad \text{(II)}$$

$$B = (1.0 - s) * [(1.0 - t) * \boldsymbol{B_0} + t * \boldsymbol{B_1}] + s * [(1.0 - t) * \boldsymbol{B_2} + t * \boldsymbol{B_3}] \quad \text{(III)}$$

$$A = (1.0 - s) * [(1.0 - t) * \boldsymbol{A_0} + t * \boldsymbol{A_1}] + s * [(1.0 - t) * \boldsymbol{A_2} + t * \boldsymbol{A_3}] \quad \text{(IV)}$$

In the implementation, formulas (I) to (IV) are rewritten as follows to minimize the number of multiplications:

$$R = uv0 * \boldsymbol{R_0} + uv1 * \boldsymbol{R_1} + uv2 * \boldsymbol{R_2} + uv3 * \boldsymbol{R_3} \quad \text{(V)}$$

$$G = uv0 * \boldsymbol{G_0} + uv1 * \boldsymbol{G_1} + uv2 * \boldsymbol{G_2} + uv3 * \boldsymbol{G_3} \quad \text{(VI)}$$

$$B = uv0 * \boldsymbol{B_0} + uv1 * \boldsymbol{B_1} + uv2 * \boldsymbol{B_2} + uv3 * \boldsymbol{B_3} \quad \text{(VII)}$$

$$A = uv0 * \boldsymbol{A_0} + uv1 * \boldsymbol{A_1} + uv2 * \boldsymbol{A_2} + uv3 * \boldsymbol{A_3} \quad \text{(VIII)}$$

where

uv0 = (1.0 – s) * (1.0 - t )

uv1 = (1.0 – s) * t

uv2 = (1.0 - t ) *s

uv3 = s*t

## 2.1 Applications for an Anti-Aliasing Algorithm

In 2D/3D computer graphics applications, anti-aliasing is a technique used to remove distortion, staircasing or jagged edges of an image created by texture mapping. The anti-aliasing technique is also used in image processing applications to reconstruct smooth images. Bilinear filtering is a sophisticated technique that averages the color values of the four adjacent pixels, thus it is fairly effective in reducing aliasing. Many applications, like games, require this level of filtering to generate a good quality of animation.

## 2.2 Implementing the Antialising Algorithm

A non-optimized version and an optimized version of Streaming SIMD Extensions implementations of anti-aliasing with supersampling are included in Sections 6.1 and 6.2, respectively.

### 2.2.1 Design Decisions

Before implementing the algorithm using Streaming SIMD Extensions, two factors must be considered:

1. Is floating point computation necessary in the algorithm?

   Although the data types of the components of the input and output pixels are bytes, the interpolations are floating point operations, because the weight factors *uv0, uv1, uv2* and *uv3* are floating point numbers. To avoid loss of accuracy, the four components of the pixel need to be converted from bytes to floats before operating on them. Therefore, floating-point computations are necessary to implement the interpolations.

2. Can the data be processed in parallel?

   Examining the interpolation formulas, we find that the same operations are applied to each of the four components of the pixels. The Streaming SIMD Extensions allow operations to occur on four floating-point numbers simultaneously. This provides the needed parallelism.

### 2.2.2 Flow of the Code

The assembly code primarily consists of a data setup section and a loop. The pixel values of the image are calculated and assigned in the iterations of the loop. The data setup section is less significant; therefore, the discussion about implementing this algorithm concentrates on the loop iteration. To simplify the analysis, the iteration is partitioned into four main sections.

#### 2.2.2.1 Interpolation Weight Factors Setup

As discussed in Section 2, the weight factors, *t* and *s*, are the decimal parts of the floating-point numbers u and v that are stored in the UV lookup table. To operate on the RGBA data in parallel, both *t* and *s* need to be stored as packed floats in Pentium® III xmm registers. At the beginning of the loop iteration, the floating-point numbers u and v are loaded into an xmm register from the lookup table. The

integer parts of u and v are extracted to an MMX™ technology register by using `cvtps2pi` instruction. The fractional parts of u and v are calculated by first converting the integer parts ([u],[v]) that have been stored in an MMX technology register to two floats in an xmm register, followed by a floating point subtraction using the `subps` command. Figure 2 shows how to calculate the weight factor *t* and store it as packed floats in an xmm register. The weight factor *s* is calculated and stored in a similar way.

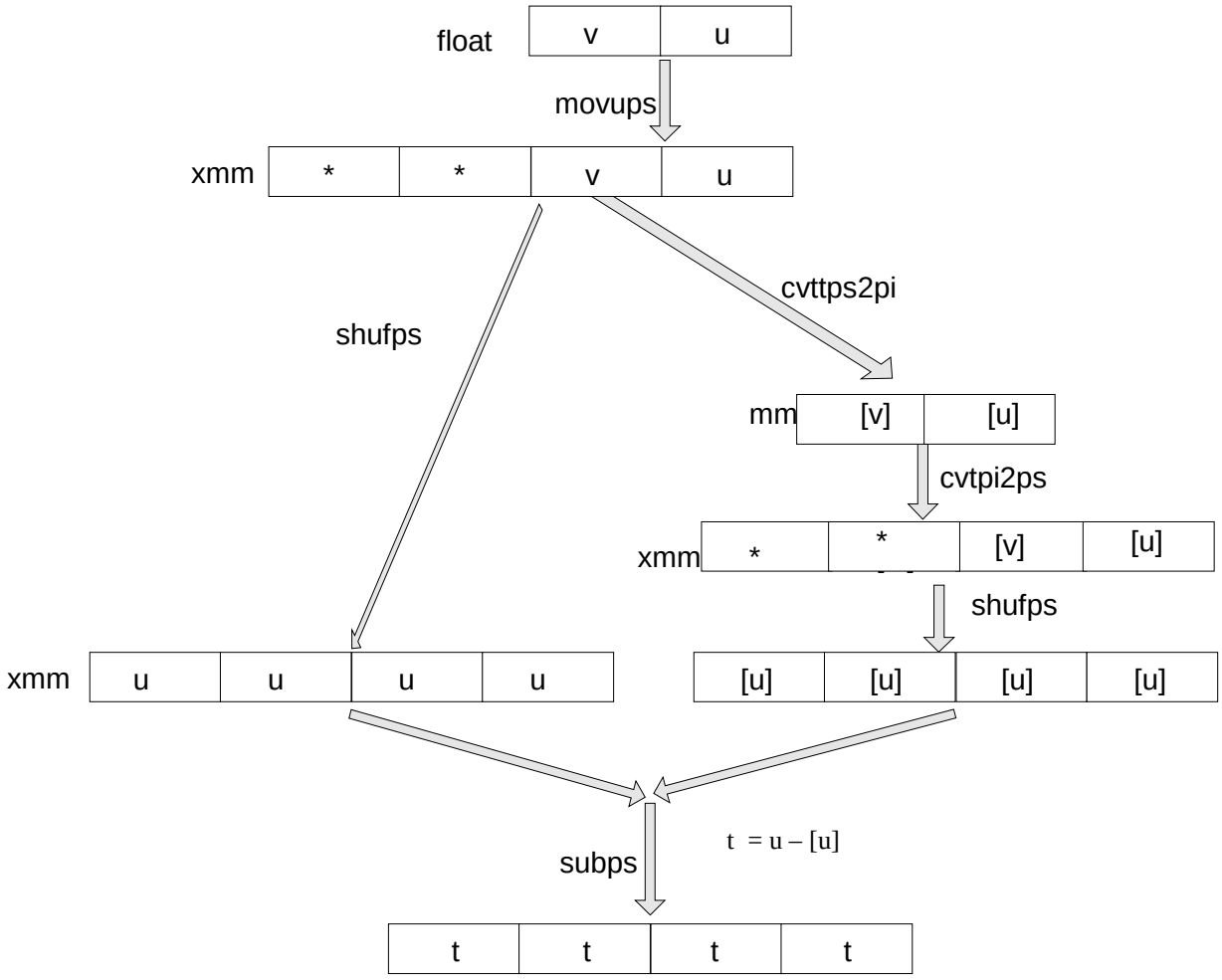


**Figure 2: Calculations of the weight factor t**

### 2.2.2.2 Conversion of the four components of a pixel from bytes to floats

Recall from the design decisions of Section 2.2.2 that it is necessary to convert the RGBA components of the superpixels from bytes to floats before operating on the data. The components of the superpixels are stored as bytes; therefore, an integer SIMD instruction is used to read in RGBA data as packed bytes. The packed bytes are converted to packed DWORDs using the `punpcklbw` and `punpcklwd` MMX instructions. Then the packed DWORDs are converted to packed floats with the `cvtpi2ps` instruction. Figure 3 illustrates the process of converting the four components of a pixel from packed bytes to packed floats, and storing the results in an xmm register.

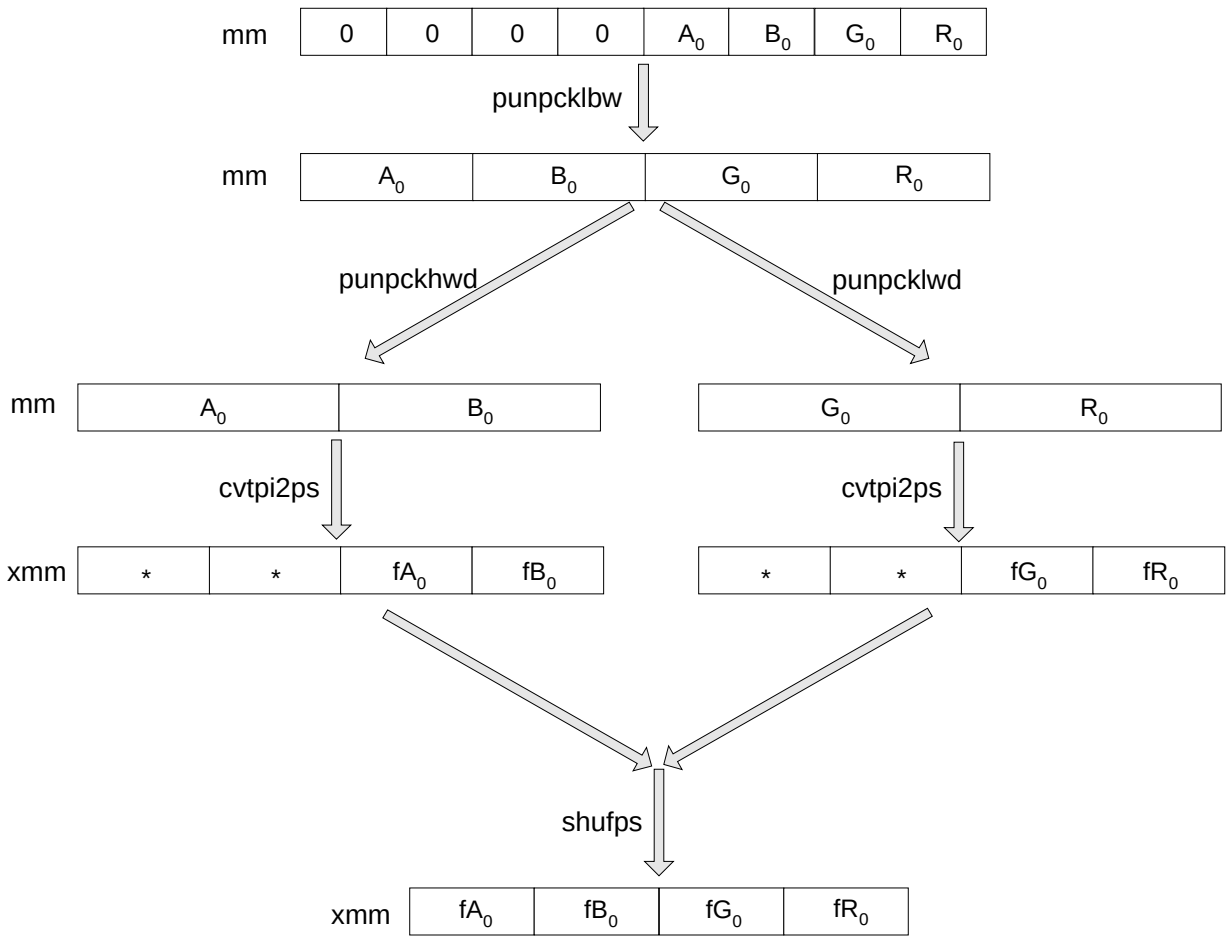


**Figure 3: Conversion of RGBA from packed bytes to packed floats**

### 2.2.2.3 Operations of the bilinear interpolation

After the RBGA data of the four superpixels is stored as packed floats in xmm registers, it is ready for interpolation. To implement formulas (V) through (VIII) in parallel, the weight factors $uv_0$, $uv_1$, $uv_2$ and $uv_3$ need to be stored as packed floats in an xmm register. Figure 4 shows the flow of the calculation of weight factor $uv_3$, and how the four components ($fR_3$,$fG_3$,$fB_3$,$fA_3$) of a superpixel are multiplied by $uv_3$ in parallel. Figure 5 shows how the four components of the weighted RGBA values are summed in parallel.

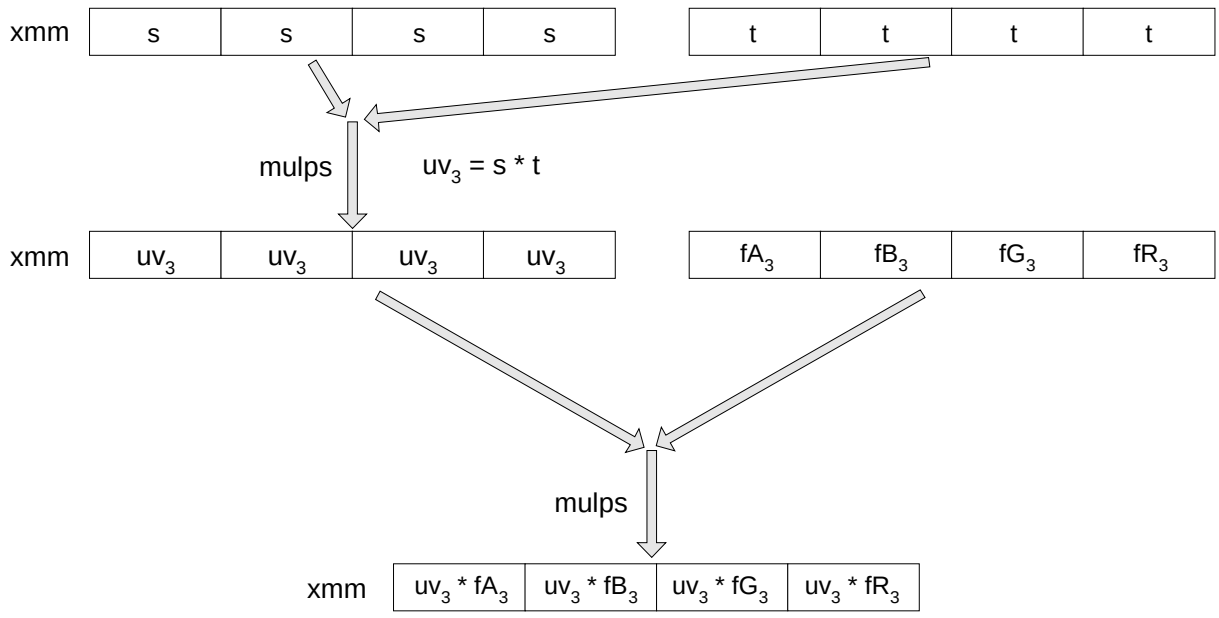


**Figure 4: Four components of the superpixel are multiplied by weight factors in parallel**

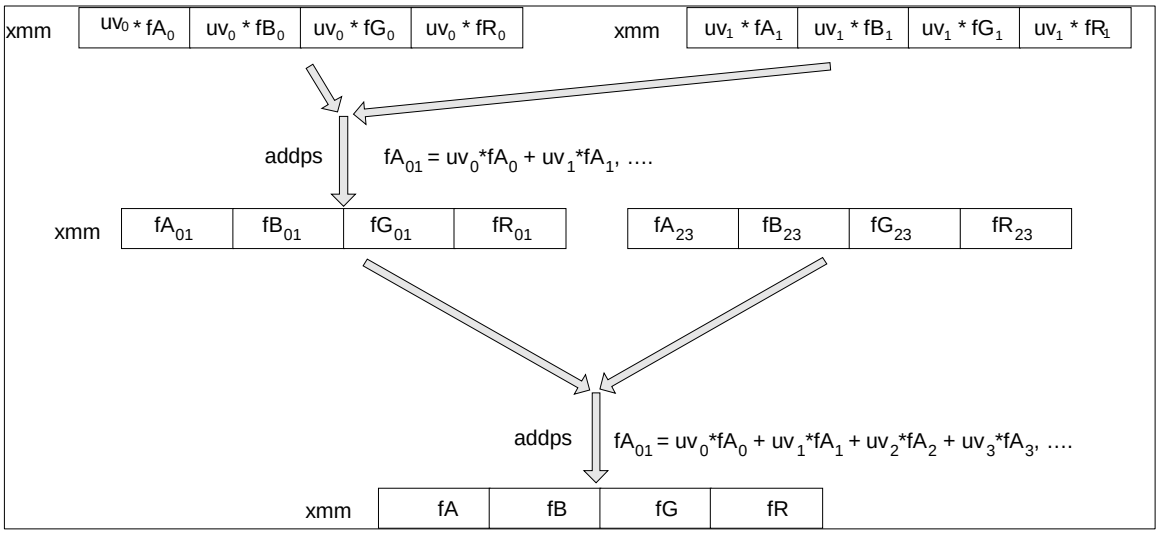


**Figure 5: Four components of the weighted RGBA values are summed up in parallel**

### 2.2.2.4 Conversion of the four components of a pixel from floats to bytes

After the interpolations, the RGBA output data is stored as packed floats in an xmm register. Before the output data is written to memory, the RGBA data needs to be converted to bytes. The process of converting the RGBA data from floats to bytes is actually the reverse of converting the RGBA from bytes to floats (refer to Figure 3).

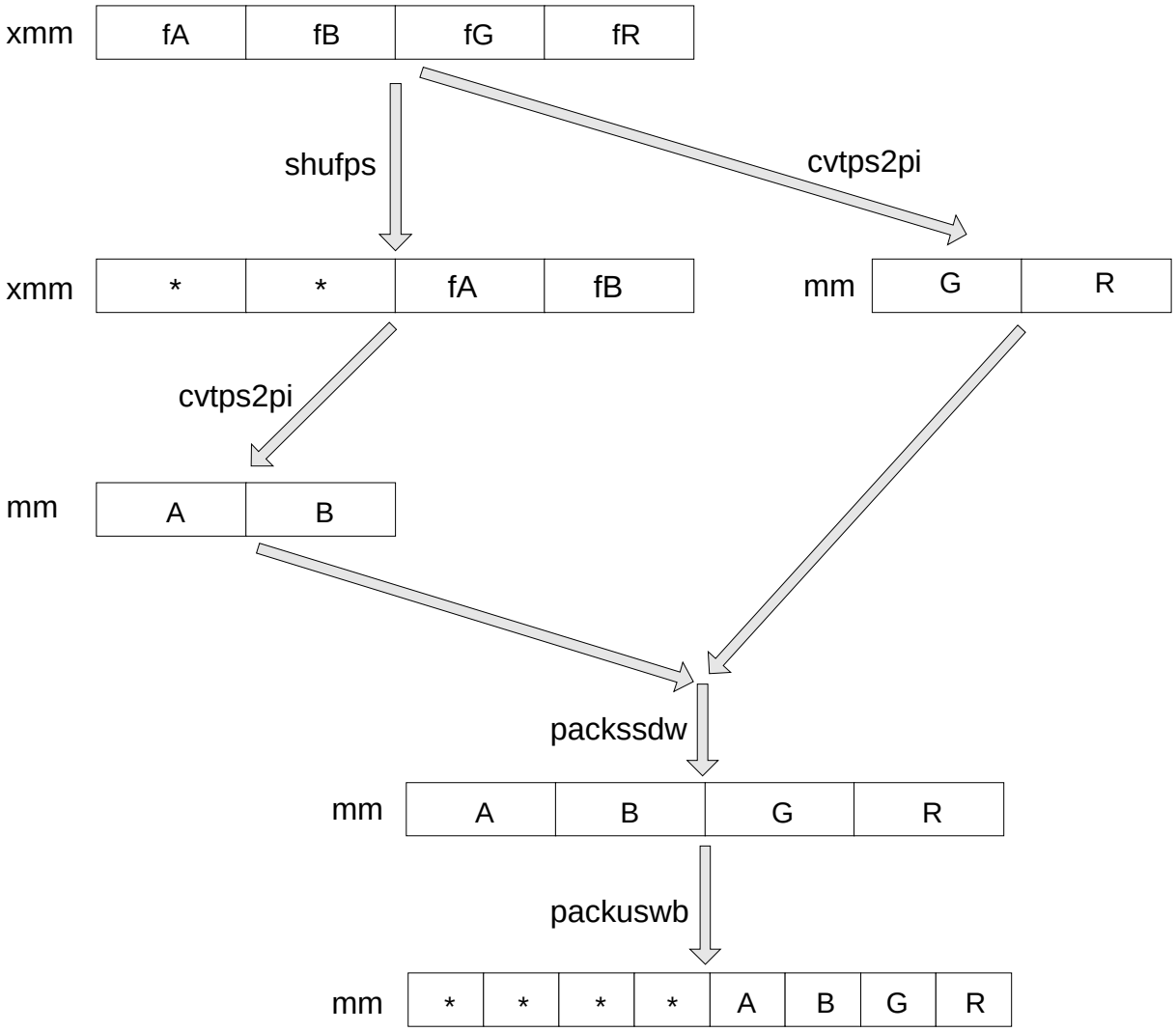


**Figure 6: Conversion of RGBA from packed floats to packed bytes**

# 3 Performance

1.

## 3.1 Gains/Improvements

The implementation of antialiasing with bilinear filtering algorithm using Streaming SIMD Extensions provides a significant speedup when compared to the C implementation. Performance gains of the Streaming SIMD Extensions are due to their SIMD nature, which allows the additions and multiplication operating on the four components of a pixel to be performed in parallel. In the C implementation, each component of a pixel is calculated individually. The Streaming SIMD Extensions and MMX technology also help to reduce memory accesses. In the Streaming SIMD Extensions implementation, four components of a pixel are loaded or stored to memory with a single instruction.

## 3.2 Considerations

Using the prefetch instruction with the assumption that the data of the UV lookup table and superpixels is not in cache speeds performance. In the optimized code, one prefetch instruction is used in the loop to pre-load the next cache line of uv data. However, prefetching the superpixel data is not practical. The memory location of the superpixel data is not predictable within the loop. By contrast, the accesses of the uv data are sequential; therefore, the prefetching provides a benefit in this case.

# 4 Conclusion

Anti-aliasing with bilinear filtering is a commonly used technique to generate a smooth image in 2D/3D computer graphics and image processing applications. A disadvantage of using bilinear filtering is that it is computationally expensive. It takes twenty floating-point multiplications to calculate one pixel value (refer to Formulas V through VIII in Section 2). While bilinear filtering is fairly effective in reducing aliasing, performance is degraded. In a game application, for example, the animations will be slowed down. This disadvantage may keep software developers, especially game developers, from using bilinear filtering. Due to the floating-point nature of the bilinear filtering algorithm, MMX technology cannot be applied to the algorithm without changing the floating-point nature of the weighted factors. With Streaming SIMD Extensions, it is possible for software developers to achieve enhanced performance for bilinear filtering. Based on the performance data presented in Section 3, the implementation of the antialiasing algorithm discussed in this paper using Streaming SIMD Extensions offers significant improvement over a C implementation.

# 5 C Coding Example

```
//*************************************************************************
// void BilinearFilter(long imageWidth,long imageHeight,
//      long lineoffset,
//      float *uv_table,
//      Pixel* SuperPixels,
//      Pixel* imagePixs)
//
// This function calculates the pixel values (red,green,blue,alpha)
// of a display image by a bilinear interpolation to a supersampling
// pixels.
//
// Inputs:
//   long imageHeight
//       Number of rows of the display image.
//   long imageWidth
//       Number of pixels in each row of the display image.
//   Pixel* SuperPixels
//       Pointer pointing to the Superpixel buffer.
//   long lineoffset
//       Number of pixels in each row of the input SuperPixels.
```

```
//   float *uv_table
//      u and v coordinates of the pixels of the display image
//       in the UV domain of the supersampling space.
//
// Output:
//   Pixel* ImagePix
//       Pointer pointing to the output pixel buffer of the display image.
//
// data structure of Pixel:
//   typedef struct pixel
//   {
//     unsigned char red, green, blue, alpha;
//   }Pixel;
//
//*********************************************************************

void BilinearFilter(long imageWidth, long imageHeight,
                    long lineoffset, float *uv_table,
                    Pixel* SuperPixels, Pixel* imagePixs)
{
  int  i;
  int  u_index, v_index;
  long num_pixs = imageWidth * imageHeight;
  long offset, offset2;
  float u,v;
  float uv0,uv1,uv2,uv3;

  for(i = 0; i < num_pixs; i++){

// Each iteration will calculate the RGBA values for a
// pixel of the display image by a bilinear interpolation
// to four super pixels

// Calculate the row and column of the correspondent superpixels
    u_index = (int) *uv_table[0];    // row of the superpixel
    v_index = (int) *uv_table[1];    // column of the superpixel

// Calculate the weights of the interpolations
    u = uv_table[0] - u_index;       // weight factor in u direction
    v = uv_table[1] - v_index;       // weight factor in v direction

// weights of the four super pixels
    uv3 = u*v;
```

```
    uv2 = v - uv3;
    uv1 = u - uv3;
    uv0 = 1.0f - u - v + uv3;

// offset address of the superpixel in first and second rows
    offset = v_index*lineoffset + u_index;
    offset2 = offset + lineoffset;

    imagePixs->red = (unsigned char)
         (uv0*pixels[offset].red + uv1*pixels[offset+1].red +
          uv2*pixels[offset2].red + uv3*pixels[offset2+1].red + 0.5);
    imagePixs->green = (unsigned char)
         (uv0*pixels[offset].green + uv1*pixels[offset+1].green+
          uv2*pixels[offset2].green + uv3*pixels[offset2+1].green + 0.5);
    imagePixs->blue  = (unsigned char)
         (uv0*pixels[offset].blue + uv1*pixels[offset+1].blue +
          uv2*pixels[offset2].blue + uv3*pixels[offset2+1].blue + 0.5);
    imagePixs->alpha = (unsigned char)
         (uv0*pixels[offset].alpha + uv1*pixels[offset+1].alpha +
          uv2*pixels[offset2].alpha + uv3*pixels[offset2+1].alpha + 0.5);

    imagePixs++;                    // Move to next pixel
  }
  uv_table += 2;

}
```

# 6 Streaming SIMD Extensions Assembly Code Example

## 6.1 Unoptimized Version

```
//*************************************************************
// void BilinearFilter_XMM(long imageWidth, long imageHeight,
//                         long lineoffset, float *uv_table,
//                         Pixel* SuperPixs, Pixel* ImagePixs)
//
// This function calculates the pixel values (red,green,blue,alpha)
// of a display image by a bilinear interpolation to a supersampling
// pixels.
//
```

```
//Inputs:
//  long image Width
//     Number of pixels in each row of the display image.
//  long image Height
//     Number of rows of the display image.
//  Pixel* SuperPixels
//     Pointer pointing to the Superpixel buffer.
//  long lineoffset
//     Number of bytes in each row of the input SuperPixels.
//   float *uv_table
//      u and v coordinates of the pixels of the display image
//      in the UV domain of the supersampling space.
//
//Output:
//  Pixel* ImagePix
//     Pointer pointing to the output pixel buffer of the image.
//
//Data structure of Pixel:
//  typedef struct pixel
//  {
//    unsigned char red, green, blue, alpha;
//  }Pixel;
//
//*********************************************************************
#define   s_01010101     0x55      // 85
#define   s_1110         0xE       // 14
#define   s_1000100      0x44      // 68

static float four_ones[4] = {1.0f, 1.0f, 1.0f, 1.0f};

void mmx_BilinearFilter(long imageWidth, long imageHeight,
                        long lineoffset, float *uv_table,
                        Pixel* SuperPixs, Pixel* ImagePixs)
{
  float  f_one = 1.0;

  __asm{

    ; Initialization of data
    mov         ecx, imageWidth          ; image width -> ecx
    mov         eax, imageHeight         ; image height -> eax
    mov         esi, SuperPixs           ; load superpixel ptr -> esi
    nul         ecx                      ; eax = width * height
```

```
    mov         edi, ImagePixs          ; load image pixel ptr -> edi
    mov         ecx, 4
    mov         edx, lineoffset         ; line offset of sample pixel
    mov         ebx, uv_table           ; u,v lookup table ptr -> ebx
    movd        mm0, edx                ; [*|lineoffset] -> mm0
    movd        mm5, ecx                 ; [*|4] -> mm5
    pxor        mm6,mm6                  ; clear mm6
    punpcklwd   mm5,mm0                  ; [*|*|lineoffset|4] -> mm5

NextPix:
  ; Calculate the RGBA values of a pixel by a bilinear
  ; interpolation to four neighbor supersampling pixels

  ; 1   Set up the interpolation weight factor and superpixel offset
  ; 1.1 Calculate the u,v index of neighbour supersampling pixels
  ;      from uv_table
  ; 1.2 Calculate the weight factors  tu,su,tv,sv
  ;      where   tu = u - [u], tv = v - [v],
  ;              su = 1.0 - tu, sv = 1.0 - tv
  ; 1.3 Calculate the offset address of the four neighbour superpixels
  ; 2   Load the four neighbor supersampling pixels and convert the
  ;      RGBA from bytes to floats
  ; 3   Implement bilinear interpolation in two steps
  ; 4   Convert the resulted RGBA from floats to bytes and write to memory

    movups     xmm7, [ebx]          ; [v2|u2|v1|u1]-> xmm7
                                    ; load (u,v) from uv_table
    cvttps2pi   mm7, xmm7            ; SuperPix index
                                    ; [v_index1|u_index1] -> mm7
    cvtpi2ps   xmm6, mm7            ; [*|*|v_index1|u_index1] -> xmm6
    subps      xmm7, xmm6           ; [*|*|tv1|tu1] -> xmm7
                                    ; tu1=u1-u_index1,tv1=v1 v_index1
    packssdw   mm7,mm7              ; [*|*|v_index1|u_index1] -> mm7
    pmaddwd    mm7,mm5              ; [*|v_index*lineoffset+4*u_index]->mm7

  ; Calculate offset of the pixel in the first row.
    Movd       ecx, mm7             ; (|v_index*lineoffset+4*u_index)->eax

  ; Load two SuperPixels on the first row.
    Movd       mm0, [esi+ecx]       ; [0|0|0|0|a0|b0|g0|r0]->mm0
    movd       mm1, [4][esi+ecx]    ; [0|0|0|0a1|b1|g1|r1]->mm1

  ; Calculate the offset of superpixel in second row
```

```
  add        ecx, esi

; Load two SuperPixels on the second row
  movd       mm2, [ecx + edx]      ; [a3|b3|g3|r3|a2|b2|g2|r2]->mm2
  movd       mm3, [4][ecx+edx]     ; [a3|b3|g3|r3|a2|b2|g2|r2] ->mm3

; Convert the RGBA from packed bytes to packed floats.
  Punpcklbw  mm0, mm6                      ; [a0|b0|g0|r0] ->mm0
  punpcklbw  mm1, mm6                      ; [a1|b1|g1|r1] ->mm1
  movq       mm4, mm0                      ; [a0|b0|g0|r0] ->mm4
  punpckhwd  mm4, mm6                      ; [a0|b0] ->mm4
  punpcklwd  mm0, mm6                      ; [g0|r0] ->mm0
  cvtpi2ps   xmm4, mm4                     ; [*|*|fa0|fb0] ->xmm4
  cvtpi2ps   xmm0, mm0                     ; [*|*|fg0|fr0] ->xmm0
  shufps     xmm0, xmm4, shuf_1000100      ; [fa0|fb0|fg0|fr0] ->xmm0

  movq       mm7, mm1                      ; [a1|b1|g1|r1] ->mm7
  punpckhwd  mm7, mm6                      ; [a1|b1] ->mm7
  punpcklwd  mm1, mm6                      ; [g1|r1] ->mm1
  cvtpi2ps   xmm5, mm7                     ; [*|*|fa1|fb1] ->xmm5
  cvtpi2ps   xmm1, mm1                     ; [*|*|fg1|fr1] ->xmm1
  shufps     xmm1, xmm5, shuf_1000100      ; [fa1|fb1|fg1|fr1] ->xmm1

  punpcklbw  mm2, mm6                      ; [a2|b2|g2|r2] ->mm2
  punpcklbw  mm3, mm6                      ; [a3|b3|g3|r3] ->mm3
  movq       mm4, mm2                      ; [a2|b2|g2|r2] ->mm4
  punpckhwd  mm4, mm6                      ; [a2|b2] ->mm4
  punpcklwd  mm2, mm6                      ; [g2|r0] ->mm2
  cvtpi2ps   xmm4, mm4                     ; [*|*|fa2|fb2] ->xmm4
  cvtpi2ps   xmm2, mm2                     ; [*|*|fg2|fr2] ->xmm2
  shufps     xmm2, xmm4, shuf_1000100      ; [fa2|fb2|fg2|fr2]->xmm2

  movq       mm7, mm3                      ; [a3|b3|g3|r3] ->mm7
  punpckhwd  mm7, mm6                      ; [a3|b3] ->mm7
  punpcklwd  mm3, mm6                      ; [g3|r3] ->mm1
  cvtpi2ps   xmm5, mm7                     ; [*|*|fa3|fb3] ->xmm5
  cvtpi2ps   xmm3, mm3                     ; [*|*|fg3|fr3] ->xmm3
  shufps     xmm3, xmm5, shuf_1000100      ; [fa3|fb3|fg3|fr3]->xmm3

; bilinear interpolation
; set up weights uv0,uv1,uv2,uv3
  movaps     xmm6, xmm7                    ; [*,*,v,u] -> xmm6
  shufps     xmm7, xmm7, s_01010101        ; [v,v,v,v] -> xmm7
```

```
    shufps      xmm6, xmm6, 0                   ; [u,u,u,u] -> xmm6
    movaps      xmm5, xmm7                      ; [v,v,v,v] -> xmm5
    movaps      xmm4, xmm6                      ; [u,u,u,u] -> xmm4
    mulps       xmm7, xmm6          ; [uv3,uv3,uv3,uv3] -> xmm7, uv3 = u*v
    subps       xmm6, xmm7          ; [uv1,uv1,uv1,uv1] ->xmm6, uv1 = u-u*v
    mulps       xmm3, xmm7          ; [uv3*fa3,uv3*fb3,uv3*fg3,uv3*fr3]->xmm3
    mulps       xmm1, xmm6          ; [uv1*fa1,uv1*fb1,uv1*fg1,uv1*fr1]->xmm1
    addps       xmm6, xmm5          ; [u+v-u*v,u+v-u*v,u+v-u*v,u+v-u*v]->xmm6
    subps       xmm5, xmm7          ; [uv2,uv2,uv2,uv2] -> xmm5, uv2 = v-u*v
    movaps      xmm7, four_ones     ; [1.0,1.0,1.0,1.0] -> xmm7
    addps       xmm1, xmm3      ;[uv1*fa1+uv3*fa3,.,.;uv1*fr1+uv3*fr3]->xmm3
    subps       xmm7, xmm6      ;[uv0,uv0,uv0,uv0] -> xmm7,uv0 = 1.0-u-v+u*v
    mulps       xmm2, xmm5      ;[uv2*fa2,uv2*fb2,.,.,uv2*fg2,uv2*fr2]->xmm2
    mulps       xmm0, xmm7      ;[uv0*fa0,uv0*fb0,uv0*fg0,uv0*fa0]->xmm0
    addps       xmm2, xmm1      ;[uv1*fa1+uv2*fa2, uv3*fa3,..,..,
                                ; uv1*fr1+uv2*fr2+uv3*fr3] ->xmm2

    addps       xmm0, xmm2      ; [fA,fB,fG,fR] - >mm0
                                ; fA = uv0*fa0 + uv1*fa1 + uv2*fa2 + uv3*fa3
                                ;
                                ; fR = uv0*fr0 + uv1*fr1 + uv2*fr2 + uv3*fr3


  ; Convert the RGBA from packed float to packed bytes
    cvtps2pi    mm0, xmm0                   ; [G|RE] ->mm0
    shufps      xmm0,xmm0,shuf_1110         ; [*|*|A|B]-> xmm1
    cvtps2pi    mm1, xmm0                   ; [A|B] ->mm1
    packssdw    mm0, mm1                    ; [A|B|G|R] ->mm0
    packuswb    mm0, mm0                    ; [*|*|*|*|A|G|B|R] ->mm0

  ; Write the RGBA output to memory
    movd        [edi], mm0
    add         ebx, 8                      ; move to next UV
    add         edi, 4                      ; move to next pixel
    dec         eax                         ; decrease the pixel counter
    jnz         NextPix
    emms
  }

}/* BilinearFilter_XMM()  */
```

## 6.2 Optimized Version

```
static float four_ones[4] = {1.0f, 1.0f, 1.0f, 1.0f};

void BilinearFilter_XMM(long imageWidth, long imageHeight,
                        long lineoffset, float *uv_table,
                        Pixel* SuperSamplePixs, Pixel* ImagePixs)
{
  __asm{
  ; Initialization of data
    mov        ecx, imageWidth          ; image width -> ecx
    mov        eax, imageHeight         ; image height -> eax
    mov        esi, SuperSamplePixs     ; load supperpixel ptr -> esi
    mul        ecx                      ; eax = width * height
    mov        edi, ImagePixs           ; load image pixel ptr -> edi
    pxor       mm6,mm6                  ; clear mm6
    mov        edx, lineoffset          ; line offset of sample pixel
    mov        ecx, 4
    movd       mm0, edx                 ; [*|lineoffset] -> mm0
    movd       mm5, ecx                 ; [*|4] -> mm5
    punpcklwd  mm5,mm0                  ; [*|*|lineoffset|4] -> mm5
    mov        ebx, uv_table            ; u,v lookup table ptr -> ebx

NextPix:
  ; Calculate the RGBA values of a pixel by bilinear
  ; interpolation to four neighbor supersampling pixels

  ; 1   Set up the interpolation weight factor and superpixel offset
  ; 1.1 Calculate the u,v index of neighbour supersampling pixels
  ;      from uv_table
  ; 1.2 Calculate the weight factors  tu,su,tv,sv.
  ;      where         tu = u - [u], tv = v - [v],
  ;                    su = 1.0 - tu, sv = 1.0 - tv
  ; 1.3 Calculate the offset address of the four neighbour superpixels
  ; 2   Load the four neighbor supersampling pixels and convert the
  ;      RGBA from bytes to floats
  ; 3   Implement bilinear interpolation
  ; 4   Convert the RGBA from floats to bytes and write to memory

    movups      xmm7, [ebx]        ; load (u,v) from uv_table
                                   ; [v2|u2|v1|u1] -> xmm7
    prefetcht0  [ebx + 32]         ; Give the processor a hint to fetch
```

```
                                ; the next cache line of data from
                                ; uv_table
cvttps2pi    mm7, xmm7           ; convert ot index of Supersample,
                                ; [v_index1,u_index1] -> mm7
cvtpi2ps    xmm6, mm7           ; [*,*,v_index1,u_index1] -> xmm6
packssdw    mm7,mm7             ; [*,*,v_index1,u_index1] -> mm7
subps       xmm7, xmm6          ; weight factors, [*,*,v,u] -> xmm7
                                ; u = u1 - u_index1,v = v1 - v_index
pmaddwd     mm7,mm5             ; [*,v_index*lineoffset+4*u_index] -> mm7
movd        ecx, mm7            ; (v_index*lineoffset+4*u_index) ->ecx

movd        mm0, [esi+ecx]          ; [0,0,0,0,,a0,b0,g0,r0] ->mm0
movd        mm1, [4][esi+ecx]       ; [0,0,0,0,a1,b1,g1,r1] ->mm1
add         ecx, esi

punpcklbw   mm0, mm6                ; [a0,b0,g0,r0] ->mm0
movd        mm2, [ecx + edx]        ; [0,0,0,0,a2,b2,g2,r2] ->mm2
punpcklbw   mm1, mm6                ; [a1,b1,g1,r1] ->mm1
movq        mm4, mm0                ; [a0,b0,g0,r0] ->mm4

movd        mm3, [4][ecx + edx]     ; [0,0,0,0,a3,b3,g3,r3] ->mm3
punpckhwd   mm4, mm6                ; [a0,b0] ->mm4
movq        mm7, mm1                ; [a1,b1,g1,r1] ->mm7
punpcklwd   mm0, mm6                ; [g0,r0] ->mm0
cvtpi2ps    xmm4, mm4               ; [*,*,fa0,fb0] ->xmm4

movaps      xmm6, xmm7              ; [*,*,v,u] -> xmm6
punpcklbw   mm2, mm6                ; [a2,b2,g2,r2] ->mm2
cvtpi2ps    xmm0, mm0               ; [*,*,fg0,fr0] ->xmm0
punpcklbw   mm3, mm6                ; [a3,b3,g3,r3] ->mm3
shufps      xmm7, xmm7,s_01010101   ; [v,v,v,v] -> xmm7
punpckhwd   mm7, mm6                ; [a1,b1] ->mm7
shufps      xmm0, xmm4, s_1000100   ; [fa0,fb0,fg0,fr0] ->xmm0

punpcklwd   mm1, mm6                ; [g1,r1] ->mm1
movq        mm4, mm2                ; [a2,b2,g2,r2] ->mm4
cvtpi2ps    xmm5, mm7               ; [*,*,fa1,fb1] ->xmm5
punpckhwd   mm4, mm6                ; [a2,b2] ->mm4
cvtpi2ps    xmm1, mm1               ; [*,*,fg1,fr1] ->xmm1
punpcklwd   mm2, mm6                ; [g2,r2] ->mm2
shufps      xmm1, xmm5, s_1000100   ; [fa1,fb1,fg1,fr1] ->xmm1

movq        mm7, mm3                ; [a3,b3,g3,r3] ->mm7
```

```
  cvtpi2ps    xmm2, mm2              ; [*,*,fg2,fr2] ->xmm2
  punpckhwd   mm7, mm6               ; [a3,b3] ->mm7
  punpcklwd   mm3, mm6               ; [g3,r3] ->mm1
  cvtpi2ps    xmm4, mm4              ; [*,*,fa2,fb2] ->xmm4
  shufps      xmm2, xmm4, s_1000100  ; [fa2,fb2,fg2,fr2] ->xmm2

  cvtpi2ps    xmm3, mm3              ; [*,*,fg3,fr3] ->xmm3
  cvtpi2ps    xmm5, mm7              ; [*,*,fa3,fb3] ->xmm5
  shufps      xmm6, xmm6, 0          ; [u,u,u,u] -> xmm6
  shufps      xmm3, xmm5, s_1000100  ; [fa3,fb3,fg3,fr3] ->xmm3

  movaps      xmm5, xmm7           ; [v,v,v,v] -> xmm5
  mulps       xmm7, xmm6           ; [uv3,uv3,uv3,uv3] -> xmm7, uv3 = u*v
  movaps      xmm4, xmm6           ; [u,u,u,u] -> xmm4

  subps       xmm6, xmm7       ; [uv1,uv1,uv1,uv1] ->xmm6, uv1 = u-u*v
  mulps       xmm3, xmm7       ; [uv3*fa3,uv3*fb3,uv3*fg3,uv3*fr3] ->xmm3
  mulps       xmm1, xmm6       ; [uv1*fa1,uv1*fb1,uv1*fg1,uv1*fr1] ->xmm1
  addps       xmm6, xmm5       ; [u+v-u*v,u+v-u*v,u+v-u*v,u+v-u*v] -> xmm6
  subps       xmm5, xmm7       ; [uv2,uv2,uv2,uv2] -> xmm5, uv2 = v-u*v
  movaps      xmm7, four_ones  ; [1.0,1.0,1.0,1.0] -> xmm7
  addps       xmm1, xmm3     ;[uv1*fa1+uv3*fa3,.,.,uv1*fr1+uv3*fr3]->xmm3
  subps       xmm7, xmm6     ;[uv0,uv0,uv0,uv0] -> xmm7, uv0 = 1.0-u-v+u*v
  mulps       xmm2, xmm5     ; [uv2*fa2,uv2*fb2,uv2*fg2,uv2*fr2] ->xmm2
  mulps       xmm0, xmm7     ; [uv0*fa0,uv0*fb0,uv0*fg0,uv0*fa0] ->xmm0
  addps       xmm2, xmm1     ; [uv1*fa1+uv2*fa2+ uv3*fa3
                             ; ,..,..,uv1*fr1+uv2*fr2+uv3*fr3] ->xmm2
  addps       xmm0, xmm2     ; [fA,fB,fG,fR] - >mm0
                             ; fA = uv0*fa0+uv1*fa1+uv2*fa2+uv3*fa3
                             ;.
                             ; fR = uv0*fr0+uv1*fr1+uv2*fr2+uv3*fr3

; Convert packed floats to packed bytes and write to memory
  cvtps2pi    mm0, xmm0                  ; [G|R] ->mm0
  shufps      xmm0, xmm0, shuf_1110      ; [A|B] -> xmm1
  cvtps2pi    mm1, xmm0                  ; [A|B] -> mm1
  packssdw    mm0, mm1                   ; [A|B|G|R] ->mm0

  add         edi, 4
  packuswb    mm0, mm0                   ; [*|*|*|*|A|B|G|R] ->mm0
  add         ebx, 8
  movd        [edi - 4], mm0
```

```
    dec         eax
    jnz         NextPix

    emms
  }
}/* BilinearFilter_XMM()  */
```